
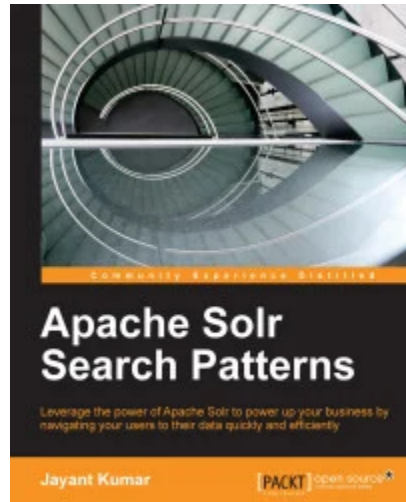


Apache Solr Search Patterns

 subscription.packtpub.com/book/data/9781783981847/9/ch09lv1sec72/fault-tolerance-and-high-availability-in-solrcloud



Chapter 9. SolrCloud

In this chapter, we will learn about SolrCloud. We will look at the architecture of SolrCloud and understand the problems it addresses. We will look at how it can be used to address scalability issues. We will also set up SolrCloud along with a separate setup for central configuration management known as ZooKeeper. We will look at the advanced sharding options available with SolrCloud, memory management issues, and monitoring options. We will also evaluate SolrCloud as a NoSQL storage system.

The major topics that will be covered in this chapter are:

- The SolrCloud architecture
- Centralized configuration
- Setting up SolrCloud
- Distributed indexing and search
- Advanced sharding with SolrCloud
- Memory management
- Monitoring

- Using SolrCloud as a NoSQL database



The SolrCloud architecture

Scaling proceeds in two ways when it comes to handling large amounts of data, horizontally or vertically. Vertical scaling deals with the problems of handling large data by adding bigger and bigger machines. Suppose a single machine which has 4 GB of RAM and 4 CPU can handle a concurrency of 100 queries per second on a data size of say 8 GB. As the amount of data increases, the amount of processing required for serving the queries also increases. Therefore, if the data size goes to 16 GB, the query concurrency that the same machine can handle will be 75 queries instead of 100. For vertical scaling, we would replace the current 4 GB + 4 CPU machine with an 8 GB + 8 CPU machine, which should again be able to serve a concurrency of 100 queries per second on a data size of 16 GB. Horizontal scaling would mean that we add another machine of the same configuration 4 GB RAM + 4 CPU to the system and divide 16 GB of data into two parts of 8 GB each. Each machine now hosts 8 GB of data and can support a concurrency of 100 queries per second. That is, a combined concurrency of 200 queries per second is obtained.

There is a limit to which a system can be scaled vertically. The largest machine available on Amazon as of now has *32 vCPUs* and *244 GB of RAM*. While it may be possible to scale a system in a vertical fashion by adding more hardware, horizontal scaling is still preferable. Maybe a year down the line, Amazon will be able to offer *64 vCPUs with 488 GB of RAM*. What if your data grows exponentially during the one-year period? The larger machine may not be able to satisfy your queries per second requirements. Horizontal scaling is cost-effective as it is possible to retain the existing hardware and add new instead of discarding the existing hardware for an upgraded or better machine. Horizontal scaling not only adds new machines providing additional computing power and memory, but it also provides additional storage. It can be made to act as a distributed system taking care of failover and high availability scenarios wherever required.

Scaling with Solr is as complex. It is possible to add hardware and scale a single Solr or a Solr setup in master-slave architecture in a vertical fashion. For vertical scaling, we will need to continually add more memory and increase the computing power and, if possible, move to SSD drives, which are expensive but a lot more efficient than normal drives. This will improve the disk IO multiple times. Since we need a master-slave architecture for high-

availability and failover scenarios, we will need to retain at least two machines, one acting as the master and the other acting as a slave and replicating Solr index data from the master.

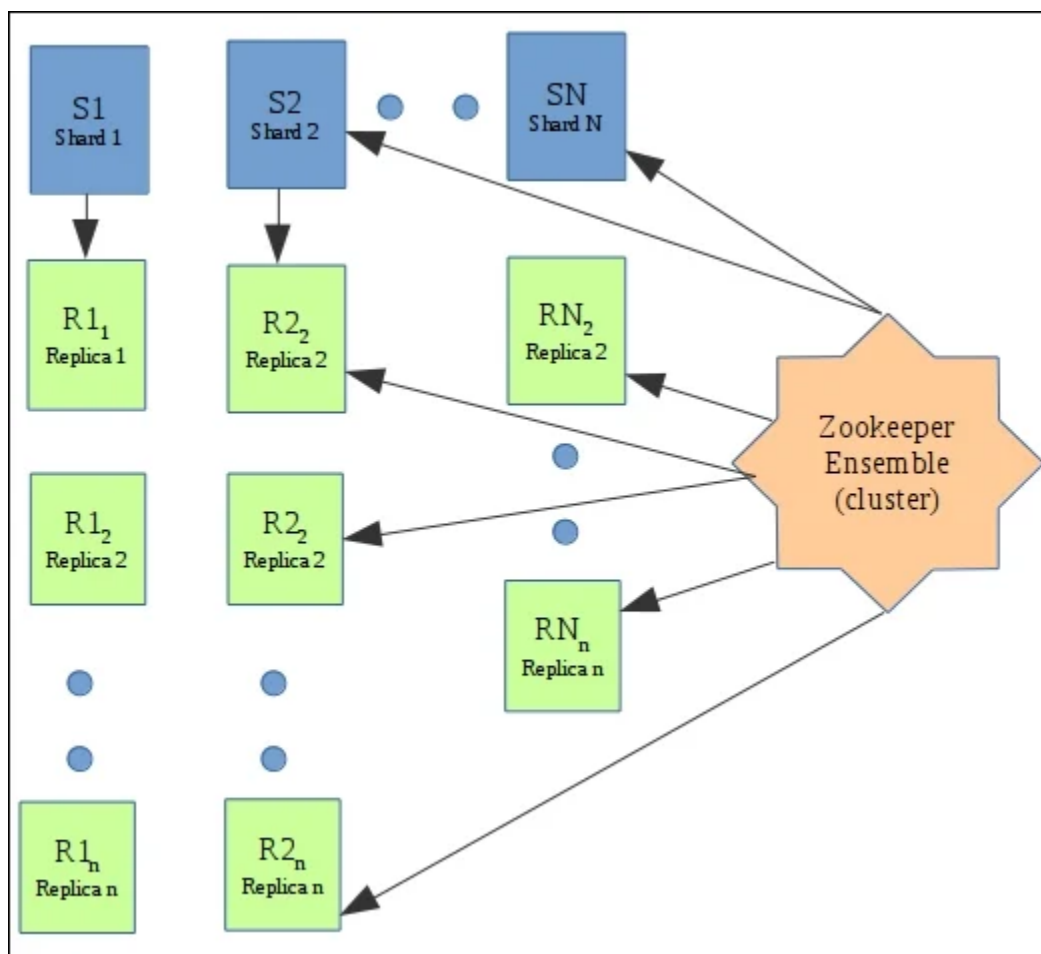
However, horizontal scaling is preferable. SolrCloud provides easy scaling in a horizontal fashion.

With SolrCloud, the complete index can be divided into **shards** and **replicas**. A shard is a part of the complete index. A replica is basically a Solr slave that reads data from the master and replicates it. A shard can have more than one replica. SolrCloud has the capability to set up a cluster of Solr servers that also provides fault tolerance and high availability in addition to distributed search and indexing.

SolrCloud offers a centralized configuration, automatic load balancing and failover for queries, and ZooKeeper integration for cluster coordination and configuration. ZooKeeper is a centralized service for maintaining configuration information for a distributed system. It is possible to have a cluster of ZooKeeper services providing high availability and failover. SolrCloud does not have a master node to allocate nodes, shards, and replicas. Instead, it uses ZooKeeper to manage these components.

With SolrCloud, we can add documents to our distributed index via any server in the cluster. The document is automatically routed to the proper shard in the cluster and indexed there. In case of a server or Solr instance going down, another shard will be elected as a leader. The searches are available near real time after indexing.

A full-scale SolrCloud setup is explained by the following architecture:



The SolrCloud architecture

The cloud consists of N shards, each of which can be on a different machine. Each shard can have n replicas, again on different machines. The configuration is managed by a separate cluster of ZooKeeper servers known as **Zookeeper ensemble**. The ZooKeeper ensemble will interact with each machine in SolrCloud, namely shard leaders and replicas.



Centralized configuration

While dealing with SolrCloud, the question that comes to mind is how any change in the schema or configuration will be propagated to the different nodes in the cluster. The ZooKeeper ensemble takes care of this.

The ZooKeeper ensemble is another cluster of servers having high availability and failover solutions built into the system. It takes care of the distribution of the schema, configuration and other files, and maintenance of the leader and replica information with regard to SolrCloud. The advantage of this is that whenever any change in the schema or configuration occurs, we need not worry about how it is propagated to all the nodes in the cluster; the ZooKeeper ensemble takes care of the propagation.

It is generally recommended to keep at least three servers for the ZooKeeper ensemble to provide for the failover scenarios. The ZooKeeper service can be run as a separate service, as nodes running SolrCloud, or in separate machines. The ZooKeeper process is lightweight and is not resource intensive.



Setting up SolrCloud

Let us set up SolrCloud. We will look at two ways of setting up SolrCloud. One setup is the ZooKeeper service running inside SolrCloud. This can be considered as a dev or a test setup that can be used for evaluating SolrCloud or for running benchmarks. Another is the production setup where SolrCloud is set up as part of the Apache Tomcat application server and the ZooKeeper ensemble as a separate service. Let us start with the test setup.

Test setup for SolrCloud

We will create a test setup of a cluster with two shards and two replicas. The Solr installation directory comes inbuilt with the packages required to run SolrCloud. There is no separate installation required. ZooKeeper is also inbuilt in the SolrCloud installation. It requires a few parameters during Solr start-up to get ZooKeeper up and running.

To start SolrCloud, perform the following steps:

- Create four copies of the example directory in the Solr installation, namely **node1** , **node2** , **node3** , and **node4** :

```
cp -r example/ node1
cp -r example/ node2
cp -r example/ node3
cp -r example/ node4
```

- To start the first node, run the following command:

```
cd node1
java -DzkRun -DnumShards=2 -Dbootstrap_confdir=./solr/collection1/conf -
Dcollection.configName=myconf -jar start.jar
```

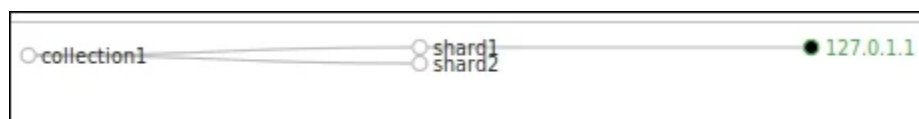
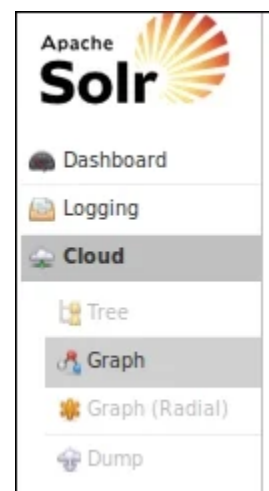
- **DzkRun** : This parameter starts the ZooKeeper server embedded in the Solr installation. This server will manage the Solr cluster configuration.
 - **DnumShards** : This parameter specifies the number of shards in SolrCloud. We have set it to 2 so that our cloud setup is configured for two shards.
 - **Dbootstrap_confdir** : This parameter instructs the ZooKeeper server to copy the configurations from this directory and distribute them across all the nodes in SolrCloud.
 - **Dcollection.configName** : This parameter specifies the name of the configuration for this SolrCloud to the ZooKeeper server.
- The output will be similar to the one shown in the following screenshot:



```
3976 [Thread-15] INFO org.apache.solr.cloud.Overseer - Update state num
Shards=2 message={
  "operation":"state",
  "state":"down",
  "base_url":"http://127.0.1.1:8983/solr",
  "core":"collection1",
  "roles":null,
  "node_name":"127.0.1.1:8983_solr",
  "shard":"shard1",
  "collection":"collection1",
  "numShards":"2",
  "core_node_name":"core_node1"}
4026 [main-EventThread] INFO org.apache.solr.common.cloud.ZkStateReader
- A cluster state change: WatchedEvent state:SyncConnected type:NodeData
Changed path:/clusterstate.json, has occurred - updating... (live nodes s
ize: 1)
```

- We can see the information that we have provided. The shard is active and is named as **shard1**. The number of shards is specified as **2**. Also, we can see that the status is updated as **live nodes size: 1**. This Solr instance will be running on port **8983** on the local server. We can open it with the following URL:
<http://localhost:8983/solr/> .

- On the left-hand panel, we can see the link for **Cloud**:
- This was not visible earlier. This option becomes visible when we start Solr with the parameters for SolrCloud. Clicking on the **Cloud** link yields the following graph:



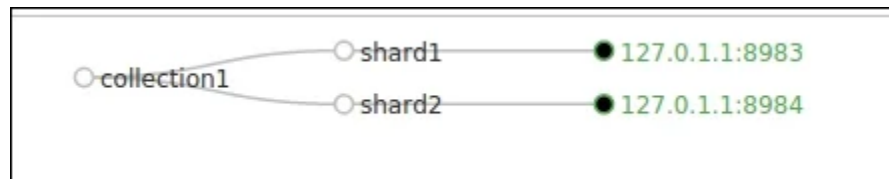
- There are also options available to have a radial view of SolrCloud. This can be seen through the **Graph (Radial)** link. We will continue to examine the SolrCloud graph to know how nodes are being added to the cloud.
- The legend for color coding of the nodes of SolrCloud is visible on the right-hand side of the interface, as shown in the following image:
- As per the legend, the current node is **Leader**. All the other functionalities of the admin interface of SolrCloud remain the same as those of the Solr admin interface.



- To start the second node, let us enter the folder called **node2** and run the following command:

```
java -Djetty.port=8984 -DzkHost=localhost:9983 -jar start.jar
```

- Djetty.port** : This parameter is required to start the Solr server on a separate port. As we are setting up all the nodes of SolrCloud on the same machine, the default port **8983** will be used for one Solr instance. Other instances of Solr will be started on separate custom ports.
 - DzkHost** : This parameter tells this instance of Solr where to find the ZooKeeper server. The port for the ZooKeeper server is Solr's port + **1000** . In our case, it is **9983** . Once the instance of Solr gets connected to the ZooKeeper server, it can get the configuration options from there. The ZooKeeper server then adds this instance of Solr to the SolrCloud cluster.
- On the terminal, we can see that the **live nodes count** has now increased to **2** . We can see the graph on the SolrCloud admin interface. It now has two shards, one running on port **8983** and the other on port **8984**:

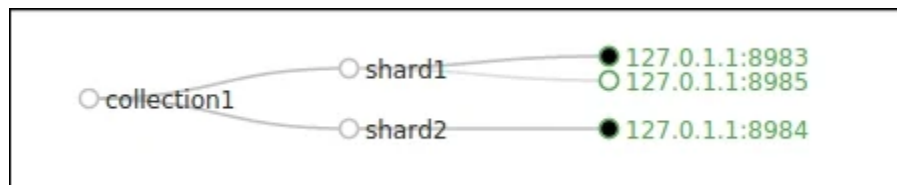


- To add more nodes, all we have to do is change the port and start another Solr instance. Let us start the third node with the following commands:

```
cd node3
```

```
java -Djetty.port=8985 -DzkHost=localhost:9983 -jar start.jar
```

- We can see that the **live nodes size** is now **3** and the SolrCloud graph has been updated:

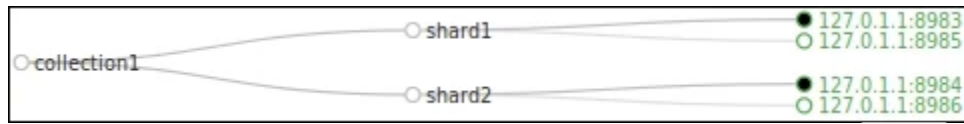


- This node is added as an active replica of the first shard.
- Let us now add the fourth node as well:

```
cd node4
```

```
java -Djetty.port=8986 -DzkHost=localhost:9983 -jar start.jar
```


- The **live nodes size** grows to **4**, and the SolrCloud graph shows that the fourth node is added as an active replica of the second shard:



This setup for SolrCloud uses a single ZooKeeper instance running on port **9983**, which is the same instance as the Solr instance running on port **8983**. This is not an ideal setup. If the first node goes offline, the entire SolrCloud setup will go offline.

Note

It is not necessary to run the SolrCloud admin interface from the first Solr running on port **8983**. We can open up the admin interface from any of the Solr servers that are part of SolrCloud. The options for cloud will be visible in all. Therefore, ideally, we can open SolrCloud from Solr servers running on ports **8984**, **8985**, and **8986** in our setup.

Setting up SolrCloud in production

The setup we saw earlier was for running SolrCloud on a single machine. This setup can be used to test out the features and functionalities of SolrCloud. For a production environment, we would want a setup that is fault tolerant and highly available. In order to have such a setup, we need at least three ZooKeeper instances. The more, the better. A minimum of three instances are required to have a fault tolerant and highly available cluster of ZooKeeper servers. As all communication between the Solr servers in SolrCloud happens via ZooKeeper, it is important to have at least two ZooKeeper instances for communication if the third instance goes down.

Setting up the Zookeeper ensemble

ZooKeeper can be downloaded from:
<http://zookeeper.apache.org/releases.html#download>.

We will set up three machines to run ZooKeeper. Let us name the machines **zoo1**, **zoo2**, and **zoo3**. The latest version of ZooKeeper is 3.4.6. Let us copy the **zookeeper tar.gz** file to the three machines and **untar** them over there. We will have a folder **zookeeper-3.4.6** on all the three machines.

On all the nodes, create a folder named **data** inside the ZooKeeper folder:

```

ubuntu@zoo1:~/zookeeper-3.4.6$ mkdir data
ubuntu@zoo2:~/zookeeper-3.4.6$ mkdir data
ubuntu@zoo3:~/zookeeper-3.4.6$ mkdir data
  
```

Each ZooKeeper server has to be given an ID. This is specified in the `myid` file in the `data` directory. We will have to create a file called `myid` inside the `data` directory on each ZooKeeper server (`zoo1` , `zoo2` , and `zoo3`) and put the ID assigned to the ZooKeeper server there. Let us assign the IDs `1` , `2` , and `3` to the ZooKeeper servers `zoo1` , `zoo2` , and `zoo3` :

```
ubuntu@zoo1:~/zookeeper-3.4.6/data$ echo 1 > myid
ubuntu@zoo2:~/zookeeper-3.4.6/data$ echo 2 > myid
ubuntu@zoo3:~/zookeeper-3.4.6/data$ echo 3 > myid
```

Copy the ZooKeeper sample configuration `zoo_sample.cfg` to `zoo.cfg` inside the `conf` folder under the `zookeeper` folder. Open the ZooKeeper configuration file `zoo.cfg` and make the following changes:

```
dataDir=/home/ubuntu/zookeeper-3.4.6/data
server.1=zoo1:2888:3888
server.2=zoo2:2888:3888
server.3=zoo3:2888:3888
```

Let the remaining setting remain as it is. We can see the `clientPort=2181` setting. This is the port where the Solr servers will connect. We have specified the `data` directory, which we just created, as also the ZooKeeper servers that are part of the ensemble, as follows:

```
server.id=host:xxxx:yyyy
```

Note the following:

- `id` : It is the ZooKeeper server ID specified in the `myid` file.
- `host` : It is the ZooKeeper server host.
- `xxxx` : It is the port used to connect to other peers for communication. A ZooKeeper server uses this port to connect followers to the leader. When a new leader arises, a follower opens a TCP connection to the leader using this port.
- `yyyy` : It is the port that is used for leader election.

Let us now start the ZooKeeper instances on all three machines. The following command starts the ZooKeeper instance:

```
ubuntu@zoo1:~/zookeeper-3.4.6$ ./bin/zkServer.sh start
```

We will have to run the command on all three machines. The output indicates that the ZooKeeper instance has started on all the three machines:

```
ubuntu@zoo1:~/zookeeper-3.4.6$ ./bin/zkServer.sh start
JMX enabled by default
Using config: /home/ubuntu/zookeeper-3.4.6/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
```

Check the logs in the `zookeeper.out` file to verify whether everything is running smoothly. ZooKeeper also comes with a client `zkCli.sh`, which can be found in the `bin` folder. In order to check whether everything is running fine, we can fire a `ruok` command via Telnet on any one of the ZooKeeper servers. If everything is running fine, we would get the output as `imok`. Another command `mntr` can be used to monitor the variables on the ZooKeeper cluster over Telnet.

We have a running ZooKeeper ensemble that we will use for setting up our SolrCloud. We will not delve into the advanced ZooKeeper settings.

Setting up Tomcat with Solr

Let's install Apache Tomcat on all the Solr servers in the `/home/ubuntu/tomcat` folder. On all the Solr servers (`solr1`, `solr2`, `solr3`, and `solr4`), start up Tomcat to check whether it is running fine:

```
ubuntu@solr1:~$ cd tomcat/
ubuntu@solr1:~/tomcat$ ./bin/startup.sh
```

We are using Tomcat version 7.0.53. Check the `catalina.out` log file inside the `tomcat/logs` folder to check whether Tomcat started successfully. We can see the following message in the logs if the start-up was successful:

```
INFO: Server startup in 2522 ms
```

Now, on one server (say `solr1`), upload all the configuration files to the ZooKeeper servers. Our Solr configuration files are located inside the `<solr_installation>/example/solr/collection1/conf` folder. To upload the files onto ZooKeeper, we will have to use the ZooKeeper client. We can copy the `zookeeper` installation folder to one of the Solr servers in order to use the ZooKeeper client. Another option is to use the ZooKeeper client inside the SolrCloud installation. For this, we will have to extract the `solr.war` file found inside the `<solr_installation>/dist` folder. Let us extract it inside a new folder `solr-war`:

```
ubuntu@solr1:~/solr-4.8.1/dist$ mkdir solr-war
ubuntu@solr1:~/solr-4.8.1/dist$ cp solr-4.8.1.war solr-war/
ubuntu@solr1:~/solr-4.8.1/dist$ cd solr-war/
ubuntu@solr1:~/solr-4.8.1/dist/solr-war$ jar -xvf solr-4.8.1.war
```

This will extract all the libraries here. We will need another library `slf4j-api` that can be found in the `<solr_installation>/dist/solrj-lib` folder:

```
ubuntu@solr1:~/solr-4.8.1/dist/solr-war/WEB-INF/lib$ cp ../../../../solrj-lib/slf4j-api-1.7.6.jar .
ubuntu@solr1:~/solr-4.8.1/dist/solr-war/WEB-INF/lib$ cp ../../../../solrj-lib/slf4j-log4j12-1.7.6.jar .
ubuntu@solr1:~/solr-4.8.1/dist/solr-war/WEB-INF/lib$ cp ../../../../solrj-lib/log4j-1.2.16.jar .
```

Now run the following command in the Solr library path `solr-war/WEB-INF/lib` to upload the Solr configuration files onto `zookeeper` on all three servers, namely `zoo1`, `zoo2`, and `zoo3`:

```
java -classpath zookeeper-3.4.6.jar:solr-core-4.8.1.jar:solr-solrj-4.8.1.jar:commons-cli-1.2.jar:slf4j-api-1.7.6.jar:commons-io-2.1.jar org.apache.solr.cloud.ZkCLI -cmd upconfig -z zoo1,zoo2,zoo3 -d ~/solr-4.8.1/example/solr/collection1/conf -n conf1
```

We have specified all the required JAR files in the `-classpath` option.

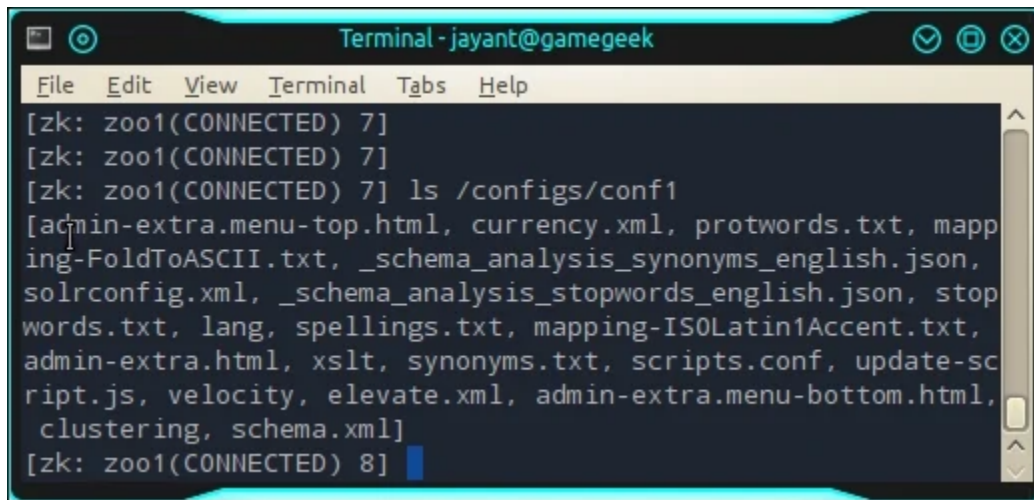
We used the zookeeper built inside the Solr cloud with the `org.apache.solr.cloud.ZkCLI` package.

- `-cmd` : This option specifies the action to be performed. In our case, we are performing the config upload action.
- `-z` : This option specifies the ZooKeeper servers along with the path (`/solr`) where the files are to be uploaded.
- `-d` : This option specifies the local directory from where the files are to be uploaded.
- `-n` : This option is the name of the config (the `solrconf` file).

In order to check whether the `configs` have been uploaded, move to machine `zoo1` and execute the following commands from the `zookeeper` folder:

```
ubuntu@zoo1:~/zookeeper-3.4.6$ ./bin/zkCli.sh -server zoo1
Connecting to zoo1
[zk: zoo1(CONNECTED) 0] ls /configs/conf1
```

This will list the config files inside the ZooKeeper servers:



```

Terminal - jayant@gamegeek
File Edit View Terminal Tabs Help
[zk: zoo1(CONNECTED) 7]
[zk: zoo1(CONNECTED) 7]
[zk: zoo1(CONNECTED) 7] ls /configs/conf1
[admin-extra.menu-top.html, currency.xml, protwords.txt, mapping-FoldToASCII.txt, _schema_analysis_synonyms_english.json, solrconfig.xml, _schema_analysis_stopwords_english.json, stopwords.txt, lang, spellings.txt, mapping-ISOLatin1Accent.txt, admin-extra.html, xslt, synonyms.txt, scripts.conf, update-script.js, velocity, elevate.xml, admin-extra.menu-bottom.html, clustering, schema.xml]
[zk: zoo1(CONNECTED) 8]

```

Let us create a separate folder to store the Solr index. On each Solr machine, create a folder named **solr-cores** inside the **home** folder:

```
ubuntu@solr1:~$ mkdir ~/solr-cores
```

Inside the folder **solr-cores**, add the following code in the **solr.xml** file. This specifies the host, port, and context along with some other parameters for ZooKeeper and the port on which Tomcat or Solr will work. The values for these variables will be supplied in the **setenv.sh** file inside the **tomcat / bin** folder:

```

<?xml version="1.0" encoding="UTF-8" ?>
<solr>
  <!-- Values are supplied from SOLR_OPTS env variable in setenv.sh -- >
  <solrcloud>
    <str name="host">${host:}</str>
    <int name="hostPort">${port:}</int>
    <str name="hostContext">${hostContext:}</str>
    <int name="zkClientTimeout">${zkClientTimeout:}</int>
    <bool name="genericCoreNodeNames">${genericCoreNodeNames:true}</bool>
  </solrcloud>

  <shardHandlerFactory name="shardHandlerFactory"
    class="HttpShardHandlerFactory">
    <int name="socketTimeout">${socketTimeout:0}</int>
    <int name="connTimeout">${connTimeout:0}</int>
  </shardHandlerFactory>
</solr>

```

Now, to set these variables, we will have to define them in the **setenv.sh** file in the **tomcat / bin** folder. Place the following code inside the **setenv.sh** file:

```

JAVA_OPTS="$JAVA_OPTS -server"
SOLR_OPTS="-Dsolr.solr.home=/home/ubuntu/solr-cores -Dhost=solr1 -Dport=8080 -
DhostContext=solr -DzkClientTimeout=20000 -DzkHost=zoo1:2181,zoo2:2181,zoo3:2181"
JAVA_OPTS="$JAVA_OPTS $SOLR_OPTS"

```

Note the following:

- `solr.solr.home` : This is the Solr home for this app instance
- `host` : The hostname for this server
- `port` : The port of this server
- `hostContext` : Tomcat webapp context name
- `zkHost` : A comma-separated list of the host and the port for the servers in the ZooKeeper ensemble
- `zkClientTimeout` : Timeout for the ZooKeeper client

Now, copy the `solr.war` file from the Solr installation into the `tomcat/webapps` folder:

```
cp ~/solr-4.8.1/dist/solr-4.8.1.war ~/tomcat/webapps/solr.war
```

Also, copy the JAR files required for logging from the `lib/ext` folder into the `tomcat/lib` folder:

```
cp -r ~/solr-4.8.1/example/lib/ext/* ~/tomcat/lib/
```

Once done, restart Tomcat. This will deploy the application `solr.war` into the `webapps` folder.

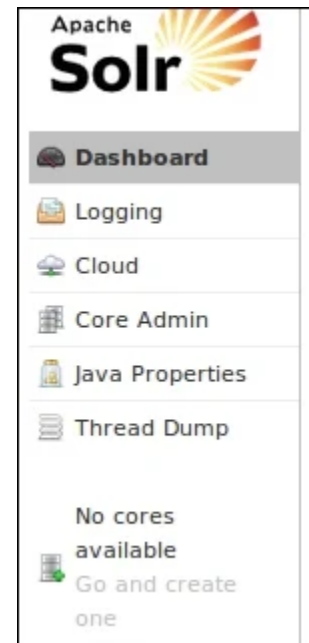
We can see Tomcat running on port `8080` on machine `solr1` and access Solr via the following URL:

```
http://solr1:8080/solr/
```

However, this does not have any cores defined.

To create the core `mycollection` in our SolrCloud, we will have to execute the `CREATE` command via the following URL:

```
http://solr1:8080/solr/admin/collections?
action=CREATE&name=mycollection&numShards=2&replicationFactor=2&maxShardsPerNode=2&c
```



Note

We require at least two running nodes to execute this command on SolrCloud. In order to start SolrCloud on multiple nodes, we can use **VirtualBox** and create multiple virtual machines on a single host.

The host name can be mapped onto the IP address in the `/etc/hosts` file on all the (virtual) machines participating in SolrCloud in the following format:

```
#<ip address> <hostname>
10.0.3.1 solr1
10.0.3.2 solr2
```

Note the following:

- `action` : `CREATE` to create the core or collection
- `name` : The name of the collection
- `numShards` : The number of shards for this collection
- `replicationFactor` : The number of replicas for each shard
- `maxShardsPerNode` : Sets a limit on the number of replicas the `CREATE` action will spread to each node
- `collection.configName` : Defines the name of the configuration to be used for this collection

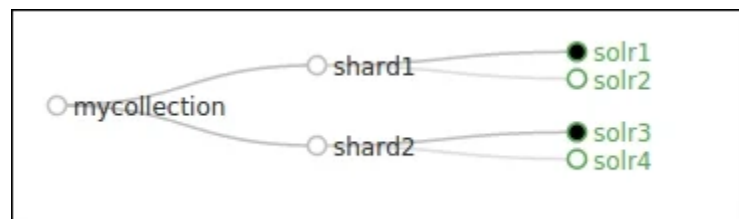
The execution of the `CREATE` action yields the following output:

We can see that two shards and two replicas are created for **mycollection** . The naming of each core is self-explanatory

```
<?xml version="1.0"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">22625</int>
  </lst>
  <lst name="success">
    <lst>
      <lst name="responseHeader">
        <int name="status">0</int>
        <int name="QTime">13807</int>
      </lst>
      <str name="core">mycollection_shard2_replica2</str>
    </lst>
    <lst>
      <lst name="responseHeader">
        <int name="status">0</int>
        <int name="QTime">13899</int>
      </lst>
      <str name="core">mycollection_shard2_replica1</str>
    </lst>
    <lst>
      <lst name="responseHeader">
        <int name="status">0</int>
        <int name="QTime">20128</int>
      </lst>
      <str name="core">mycollection_shard1_replica1</str>
    </lst>
    <lst>
      <lst name="responseHeader">
        <int name="status">0</int>
        <int name="QTime">20540</int>
      </lst>
      <str name="core">mycollection_shard1_replica2</str>
    </lst>
  </lst>
</response>
```

<collection_name>_<shard_no>_<replica_no> . We can also see the SolrCloud graph that shows the shards for the collection along with the leader and replicas:

We can see that **mycollection** has two shards, **shard1** and **shard2** . **Shard1** has leader on **solr1** and replica on **solr2** . Similarly, **Shard2** has leader on **solr3** and replica on **solr4** . The admin interface on each node of the Solr cluster will show the shard or core hosted on this node. Go to the admin interface on the node **solr1** and select the core name from the drop-down on the left-hand panel. We should be able to see the details of the index on that node:



Instance

```
CWD: /home/ubuntu/tomcat
Instance: /home/ubuntu/solr-cores/mycollection_shard1_replica1
Data: /home/ubuntu/solr-cores/mycollection_shard1_replica1/data
Index: /home/ubuntu/solr-cores/mycollection_shard1_replica1/data/index
Impl: org.apache.solr.core.NRTCachingDirectoryFactory
```


Let us also see what happened at the ZooKeeper end. Go to any of the ZooKeeper servers and connect to the ZooKeeper cluster using the `zkCli.sh` script:

```
ubuntu@zoo1:~/zookeeper-3.4.6$ ./bin/zkCli.sh -server zoo1,zoo2,zoo3
Connecting to zoo1,zoo2,zoo3
```

We can see that `mycollection` is created inside the `/collections` folder. On executing a `get`, `mycollection` is linked with the configuration `conf1` that we specified in the `collection.configName` parameter while creating the collection:

```
[zk: zoo1,zoo2,zoo3(CONNECTED) 0] get /collections/mycollection
{"configName":"conf1"}
```

We can also see the cluster configuration by getting the `clusterstate.json` file from the ZooKeeper cluster:

```
[zk: zoo1,zoo2,zoo3(CONNECTED) 1] get /clusterstate.json
{"mycollection":{
  "shards":{
    "shard1":{
      "range":"80000000-ffffffff",
      "state":"active",
      "replicas":{
        "core_node2":{
          "state":"active",
          "base_url":"http://solr1:8080/solr",
          "core":"mycollection_shard1_replica1",
          "node_name":"solr1:8080_solr",
          "leader":"true"},
        "core_node3":{
          "state":"active",
          "base_url":"http://solr2:8080/solr",
          "core":"mycollection_shard1_replica2",
          "node_name":"solr2:8080_solr"}}}},
    "shard2":{
      "range":"0-7ffffffff",
      "state":"active",
      "replicas":{
        "core_node1":{
          "state":"active",
          "base_url":"http://solr3:8080/solr",
          "core":"mycollection_shard2_replica2",
          "node_name":"solr3:8080_solr",
          "leader":"true"},
        "core_node4":{
          "state":"active",
          "base_url":"http://solr4:8080/solr",
          "core":"mycollection_shard2_replica1",
          "node_name":"solr4:8080_solr"}}}},
    "maxShardsPerNode":"2",
    "router":{"name":"compositeId"},
    "replicationFactor":"2"}}
```

This shows the complete cluster information—the name of the collection, the shards and replicas, as well as the base URLs for accessing Solr. It contains the core name and the node name. Other configuration information that we passed while creating the cluster are `maxShardsPerNode` and `replicationFactor`.



Distributed indexing and search

Now that we have SolrCloud up and running, let us see how indexing and search happen in a distributed environment. Go to the `<solr_installation>/example/exampledocs` folder where there are some sample XML files. Let us add some documents from the `hd.xml` file to SolrCloud. We will use the node `solr1` for adding documents to the index. Here we are passing the collection name in the update URL instead of the core. The output from the command execution is shown in the following snippet:

```
$ java -Durl=http://solr1:8080/solr/mycollection/update -jar post.jar hd.xml
SimplePostTool version 1.5
Posting files to base url http://solr1:8080/solr/mycollection/update using content-
type application/xml..
POSTing file hd.xml
1 files indexed.
COMMITting Solr index changes to http://solr1:8080/solr/mycollection/update..
Time spent: 0:00:21.209
```

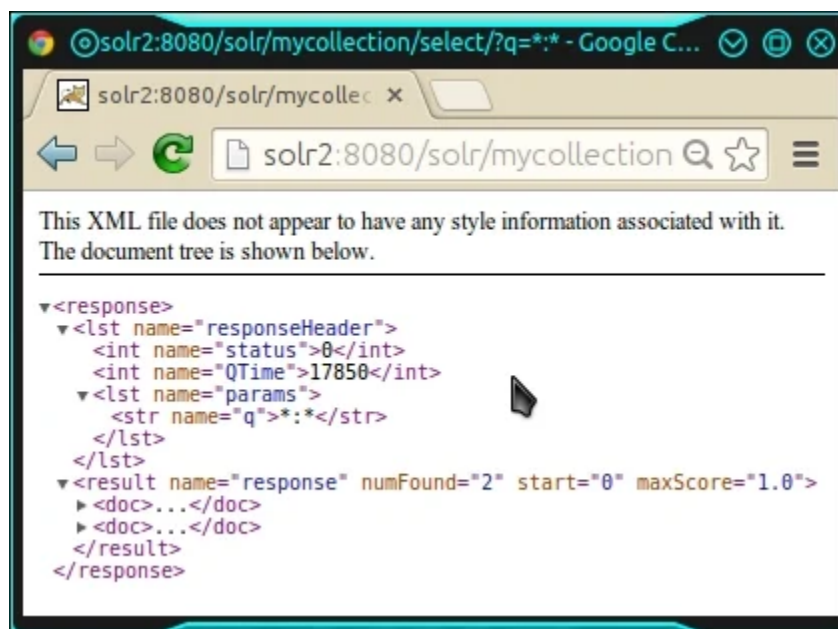
Note

Please use `localhost` instead of the `solr1` host if running on a local machine.


The documents are now committed into SolrCloud. To find the documents, perform a search on any node. Let us say we search on the node `solr2`. Execute the following query:

```
http://solr2:8080/solr/mycollection/select/?q=**
```

We can see that there are two documents in the result:



So, we actually indexed the documents via the **solr1** server in our SolrCloud and searched via the **solr2** server. The documents were indexed somewhere inside SolrCloud. In order to check where the documents went, we will have to go to each server and examine the overview of the collection. In the present case, the documents were indexed on servers **solr3** and **solr4**, which are replicas.



Dashboard

Logging

Cloud

Core Admin


Java Properties

Thread Dump

mycollection_...

Overview

Analysis

 Statistics

Last Modified: about 11 hours ago

Num Docs: 2


Max Doc: 2


Heap Memory Usage: 317


Deleted Docs: 0

Version: 3

Segment Count: 1

Optimized: 

Current: 

 Replication (Slave)

	Version	Gen	Size
Master (Searching)	1403622273385	2	5.54 KB
Master (Replicable)	1403622273385	2	-
Slave (Searching)	1403622270389	2	5.54 KB

This means that we can index documents from any shard in SolrCloud, and those documents will be routed to a server in the cloud. Similarly, we can search from any shard in the cluster and query the complete index on SolrCloud.

Later in this chapter, we will also look at how and why to send documents to a particular shard.

Let us try indexing via the other machines. Execute the following commands from the `exampledocs` folder on the remaining machines in SolrCloud:

```
java -Durl=http://solr2:8080/solr/mycollection/update -jar post.jar mem.xml
java -Durl=http://solr3:8080/solr/mycollection/update -jar post.jar vidcard.xml
java -Durl=http://solr4:8080/solr/mycollection/update -jar post.jar monitor*.xml
```

This will index the documents in the `mem.xml`, `vidcard.xml`, `monitor.xml`, and `monitor2.xml` files into SolrCloud. On searching via, say, the `solr3` machine, we can get all the documents. We indexed all nine documents, and all of them were found during a search on SolrCloud:

`http://solr3:8080/solr/mycollection/select?q=**`

This means that since we have a four-node cluster, we can now index and search via all four nodes. The nodes themselves take care of routing of the documents to their appropriate shards and indexing them. Ideally, this should result in a four-fold increase in the indexing and searching speed. However, in a real-life scenario, the number of queries per second for indexing and search would be a little less than four times. It may also depend on the amount of data in the index, the IO capabilities of the server, and the network bandwidth between the machines in the cloud.



Routing documents to a particular shard

As we have seen, SolrCloud automatically distributes documents to different shards in the index. The queries on the cloud accumulate results from all the different shards and send them back. Why then would we want to route documents to a particular shard?

Suppose that we have a huge cluster of servers as part of SolrCloud—say 100 servers—with 30 shards and 3 replicas for each shard. This gives us ample room to manage a large-scale index expanding to some terabytes of data. A query to get the documents from the index based on a criterion would go to all the 30 shards in the index to get the results. The machine on which the query is executed would accumulate results from all the 30 shards and create the final result set. This would involve huge movement of data between shards and the shard performing the merge operation on the results will have to do some heavy processing, since it would move through 30 different result sets and merge them into a single result set.

It would be better if the query hits fewer shards or probably a single shard and fetches results from that shard without performing any merge operation. This would definitely be faster and less resource and network intensive. Ideally, each query should have an identifier or a set of identifiers pointing to the shards that may contain results from the query. This may not look possible from a broad view. Nevertheless, if the indexing is performed in a fashion that the data is distributed across the shards on the basis of some identifier, it may become possible.

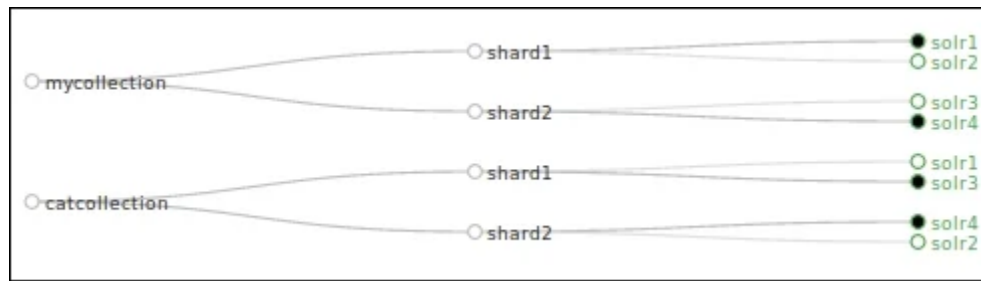
A realistic example is to have documents from different customers indexed in SolrCloud. It would make sense to route the documents on the basis of the customer ID or customer name to the different shards in the cloud. Thus, documents belonging to IBM, Samsung, Apple, and Sony can reside on different shards. While querying, we can specify a prefix in the query so that a query done by IBM hits only the shard on which IBM resides.

Let us create a separate collection in SolrCloud with a routing parameter and then index a few documents into the cloud. We will be indexing the documents in the file `docs.csv`. We will consider the category as the sharding key. Therefore, Solr will create a hash based on the category and distribute the index on the basis of that hash.

To create the collection (let us call it `catcollection`), we have to execute the following command. Note the `router.field=cat` parameter at the end of the `CREATE` command. This is how we specify a router in a collection:

```
http://solr1:8080/solr/admin/collections?
action=CREATE&name=catcollection&numShards=2&replicationFactor=2&maxShardsPerNode=2&
```

We can see that `catcollection` is created along with the previous collection `mycollection`. This collection again has two shards—`shard1` with `solr3` as **leader** and `solr1` as replica, and `shard2` with `solr4` as **leader** and `solr3` as replica:



In order to verify whether the routing that we have specified has been successful, we will have to connect to the ZooKeeper server using the `zkCli.sh` script and look at the `clusterstate.json` file:

```
ubuntu@zoo1:~/zookeeper-3.4.6$ ./bin/zkCli.sh -server zoo1,zoo2,zoo3
[zk: zoo1,zoo2,zoo3(CONNECTED) 2] get /clusterstate.json
```

The output will contain both the collections— `mycollection` and `catcollection` —along with all the configuration parameters, as follows:

```

"catcollection":{
  "shards":{
    "shard1":{
      "range":"80000000-ffffffff",
      "state":"active",
      "replicas":{
        "core_node2":{
          "state":"active",
          "base_url":"http://solr1:8080/solr",
          "core":"catcollection_shard1_replica1",
          "node_name":"solr1:8080_solr"},
        "core_node3":{
          "state":"active",
          "base_url":"http://solr3:8080/solr",
          "core":"catcollection_shard1_replica2",
          "node_name":"solr3:8080_solr",
          "leader":"true"}}},
    "shard2":{
      "range":"0-7fffffffff",
      "state":"active",
      "replicas":{
        "core_node1":{
          "state":"active",
          "base_url":"http://solr4:8080/solr",
          "core":"catcollection_shard2_replica1",
          "node_name":"solr4:8080_solr",
          "leader":"true"},
        "core_node4":{
          "state":"active",
          "base_url":"http://solr2:8080/solr",
          "core":"catcollection_shard2_replica2",
          "node_name":"solr2:8080_solr"}}}},
    "maxShardsPerNode":"2",
    "router":{
      "field":"cat",
      "name":"compositeId"},
    "replicationFactor":"2"}}

```

Here the router parameter has the field `cat` mentioned in it. Also, the range is specified in the `range` parameter for each shard. `shard1` and `shard2` will contain documents with hash IDs in the ranges `80000000-ffffffff` and `0-7fffffffff`, respectively. Therefore, when a document is marked for indexing, the router will calculate the 32 bit hash of the content in the `cat` field and route the document to the shard whose range includes the hash value of the category value:

Note

The hash values are represented in hexadecimal.

Now let us push the `docs.csv` file (available with this chapter) to the cloud and see how the documents are distributed across the shards. Execute the following command to push the documents into SolrCloud:

```
$ java -Dtype=text/csv -Durl=http://solr4:8080/solr/catcollection/update -jar
post.jar docs.csv
SimplePostTool version 1.5
Posting files to base url http://solr4:8080/solr/catcollection/update using content-
type text/csv..
POSTing file docs.csv
1 files indexed.
COMMITting Solr index changes to http://solr4:8080/solr/catcollection/update..
Time spent: 0:00:16.608
```

In order to query a particular shard, we will have to append the `shards=shard<no>` command at the end of the query. Let us see the documents in `shard1` and `shard2` :

```
http://solr1:8080/solr/catcollection/select?q=*&shards=shard1
```

We can see that documents belonging to the category `currency` are indexed on `shard1` . Similarly, on executing the query on `shard2` , we can see that the documents belonging to the categories `book` and `electronics` are indexed on `shard2` :

```
http://solr1:8080/solr/catcollection/select?
q=*&shards=shard2
```

```
<?xml version="1.0"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">0</int>
    <lst name="params">
      <str name="shards">shard1</str>
      <str name="q">*</str>
    </lst>
  </lst>
  <result name="response" numFound="4" start="0">
    <doc>
      <str name="id">USD</str>
      <arr name="cat">
        <str>currency</str>
      </arr>
      <str name="name">One Dollar</str>
      <float name="price">1.0</float>
    </doc>
  </result>
</response>
```

To send a query to a particular shard, we have to use the `shard.keys` parameter in our query. For example, to send the following query to the shard containing only books, we need to execute the following query:

```
http://solr1:8080/solr/catcollection/s?
q=martin&fl=cat,name,description&shard
```

```
<?xml version="1.0"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">422</int>
    <lst name="params">
      <str name="shards">shard2</str>
      <str name="q">*</str>
    </lst>
  </lst>
  <result name="response" numFound="7" start="0" maxScore="1.0">
    <doc>
      <str name="id">0553573403</str>
      <arr name="cat">
        <str>book</str>
      </arr>
    </doc>
  </result>
</response>
```

We can see that all the books that have the word `martin` in their descriptions become a part of the query result. If we try to execute the same query with a different shard key, say `currency`, we will not be able to get any results:

```
http://solr1:8080/solr/catcollection/select?q=martin&fl=cat,name,description&shard.
keys=currency!
```



```
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">444</int>
    <lst name="params">
      <str name="fl">cat,name,description</str>
      <str name="shard.keys">book!</str>
      <str name="q">martin</str>
    </lst>
  </lst>
  <result name="response" numFound="3" start="0" maxScore="0.4873799">
    <doc>
      <arr name="cat">
        <str>book</str>
      </arr>
      <str name="name">A Game of Thrones</str>
      <str name="description">Author: George R.R. Martin</str>
    </doc>
    <doc>
      <arr name="cat">
        <str>book</str>
      </arr>
      <str name="name">A Clash of Kings</str>
      <str name="description">Author: George R.R. Martin</str>
    </doc>
    <doc>
      <arr name="cat">
        <str>book</str>
      </arr>
      <str name="name">A Storm of Swords</str>
      <str name="description">Author: George R.R. Martin</str>
    </doc>
  </result>
</response>
```

```
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">2</int>
    <lst name="params">
      <str name="fl">cat,name,description</str>
      <str name="shard.keys">currency!</str>
      <str name="q">martin</str>
    </lst>
  </lst>
  <result name="response" numFound="0" start="0"/>
</response>
```

Adding more nodes to the SolrCloud

Let us see how we can add more nodes to SolrCloud. Create one more machine **solr5** . Copy the Tomcat folder to this machine and create the folder **solr-cores** in the **/home/ubuntu** folder. Alter the **tomcat/bin/setenv.sh** file and change the **-Dhost** parameter to match the machine's host. For **solr5** , it will be:

```
SOLR_OPTS="-Dsolr.solr.home=/home/ubuntu/solr-cores -Dhost=solr5 -Dport=8080 -DhostContext=solr -DzkClientTimeout=20000 -DzkHost=zoo1:2181,zoo2:2181,zoo3:2181"
```

Also copy the **solr.xml** file from any Solr machine to these machines inside the **solr-cores** folder. Now start Tomcat and check whether it is running by opening the following URL: **http://solr5:8080/solr** .

There are two ways to identify whether this node has been added to SolrCloud. We can check whether the admin interface on **solr5** displays the current **Cloud | Graph**.

Another way is to go to the admin | Cloud | Tree | live_nodes folder. This should contain the

name of the live nodes. **solr5** should be visible there.

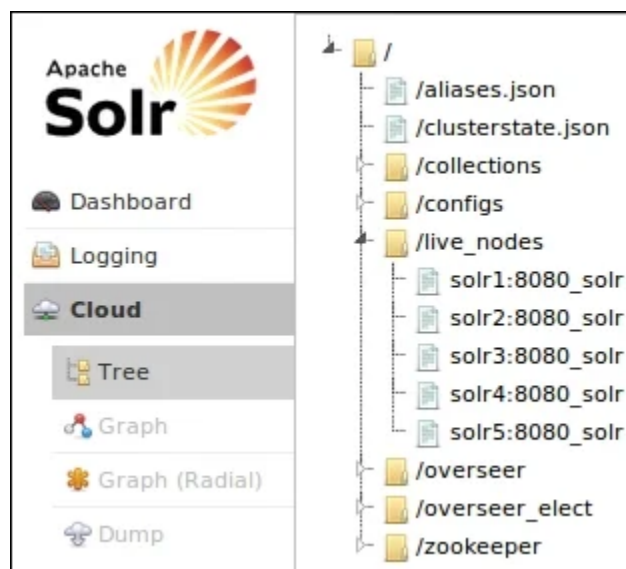
Now, let us add the node as a replica for **shard2** of **mycollection**. For this, we will have to execute the **ADDREPLICA** command on the collection API, as follows:

`http://solr5:8080/solr/admin/collections?action=ADDREPLICA&collection=mycollection`

The output from command execution will specify the name of the core that has been created:

In this command, we have specified:

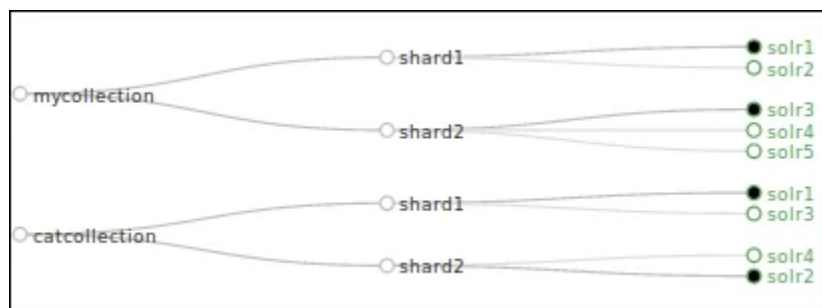
- **action=ADDREPLICA** : This is the action to be performed on the collection.
- **collection=mycollection** : This is the collection on which the action is to be performed.
- **shard=shard2** : This is the shard for which the replica is to be created.



```
<?xml version="1.0"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">7740</int>
  </lst>
  <lst name="success">
    <lst>
      <lst name="responseHeader">
        <int name="status">0</int>
        <int name="QTime">6754</int>
      </lst>
      <str name="core">mycollection_shard2_replica3</str>
    </lst>
  </lst>
</response>
```

- **node=solr5:8080_solr** : This is the node on which the replica is to be created. The name of the shard is obtained from the **live_nodes** list we saw earlier.

The cloud graph indicates that **solr5** is added as a replica of **shard2** of **mycollection**:





Fault tolerance and high availability in SolrCloud

Whenever SolrCloud is restarted, election happens again. If a particular shard that was a replica earlier comes up before the shard that was the leader, the replica shard becomes the leader and the leader shard becomes the replica. Whenever we restart the Tomcat and ZooKeeper servers for starting SolrCloud, we can expect the leaders and replicas to switch.

Let us check the availability of the cluster. We will bring down a leader node and a replica node to check whether the cluster is able to serve all the documents that we have indexed. First check the number of documents in **mycollection** :

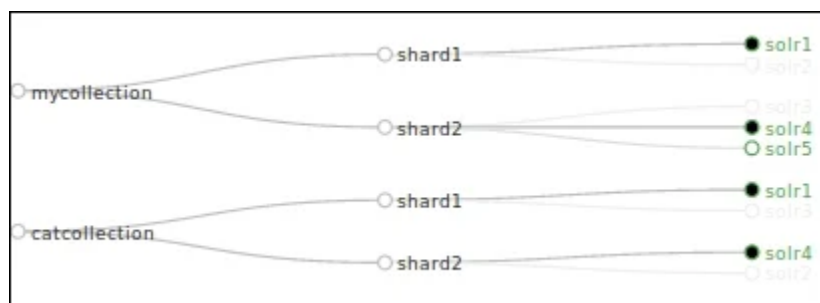
```
http://solr5:8080/solr/mycollection/select/?q=**
```

We can see that there are nine documents in the collection. Similarly, run the following query on **catcollection** :

```
http://solr5:8080/solr/catcollection/select/?q=**
```

The **catcollection** contains 11 documents.

Now let us bring down Tomcat on **solr2** and **solr3**, or simply turn off the machines. Check the SolrCloud graph:



We can see that the nodes **solr2** and **solr3** are now in the **Gone** state. Here **solr3** was the leader for **shard2** in **mycollection**. Now since it is offline, **solr4** is promoted as the leader. The replica of **shard1** that was **solr2** is not available, so **solr1** remains the leader.

Let us execute the queries we had executed earlier on both the collections. We can see that the count of documents in both the collections remains the same.

We can even add documents to SolrCloud during this time. Add the `ipod_other.xml` file from the `example/exampledocs` folder inside the Solr installation to `mycollection` on SolrCloud. Execute the following command:

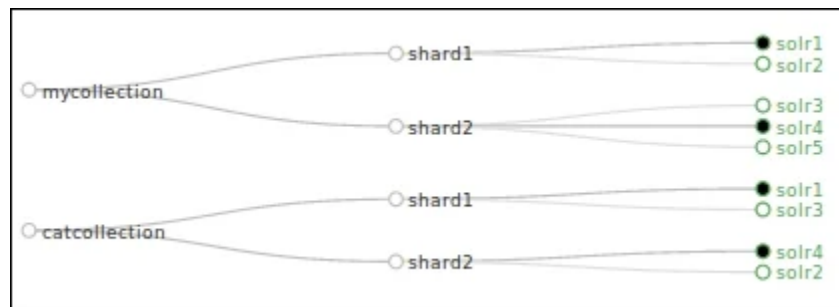
```
$ java -Durl=http://solr5:8080/solr/mycollection/update -jar post.jar ipod_other.xml
SimplePostTool version 1.5
Posting files to base url http://solr5:8080/solr/mycollection/update using content-
type application/xml..
POSTing file ipod_other.xml
1 files indexed.
COMMITting Solr index changes to http://solr5:8080/solr/mycollection/update..
Time spent: 0:00:07.978
```

Now again run the query to get the complete count from `mycollection` :

```
http://solr5:8080/solr/mycollection/select/?q=**:
```

The `mycollection` collection now contains 11 documents.

Now start up Tomcat on **solr2** and **solr3**. We can see that after a short time in recovery, both the Solr instances are now added to the cloud. For `mycollection`, **solr2** continues its role as the replica of **solr1** for **shard1**, and **solr3** becomes the replica of **solr4** for **shard2**:



The documents that were added to the cloud while any of the nodes were down are automatically replicated onto the nodes once they come back into the cloud. Therefore, as long as one of the nodes for the shard of a collection is available, the collection will remain available and will continue to support indexing and searching of documents.



Advanced sharding with SolrCloud

Let's explore some of the advanced concepts of sharding, starting with **shard splitting**.

Shard splitting

Let us say that we have created a two-shard replica looking at the current number of queries per second for a system. In future, if the number of queries per second increases to, say, twice or thrice the current value, we will need to add more shards. Now, one way is to create a separate cloud with say four shards and re-index all the documents. This is possible if the cluster is small. If we are dealing with a 50 shard cluster with more than a billion documents, re-indexing of the complete set of documents again may be expensive. For such scenarios, SolrCloud has the concept of shard splitting.

In shard splitting, a shard is divided into two new shards on the same machine. All three shards, the old one and the two new ones, remain. We can check the sanity of the shards and then delete the existing shard. Let us see a practical implementation of the same.

Before starting, let's add a few more documents into **mycollection**. Add the **books.csv** file from the **example/exampledocs** folder to **mycollection**.

```
java -Dtype=text/csv -Durl=http://solr1:8080/solr/mycollection/update -jar post.jar books.csv
```

To check the number of documents in **mycollection**, execute the following query:

```
http://solr1:8080/solr/mycollection/select?q=:*:
```

We can see that there are 35 documents currently in **mycollection**. Let us check the count of documents in each shard. Execute the following queries to find the documents in **shard1** of **mycollection**:

```
http://solr1:8080/solr/mycollection/select?q=:*&shards=shard1
```

There are **17** documents in **shard1**. Now execute the following query to find the number of documents in **shard2** of **mycollection**:

```
http://solr1:8080/solr/mycollection/select?q=:*&shards=shard2
```

We can see that there are 18 documents in **shard2**. Let us also look at how our SolrCloud graph looks. We can see that **shard1** is on **solr3** and **solr4** and **shard2** is on **solr1**, **solr2**, and **solr5**:



Now let us split **shard1** into two parts. This is done by the **SPLITSHARD** action on the collections API via the Solr admin interface. Execute the command by calling the following URL:

```
http://solr1:8080/solr/admin/collections?
action=SPLITSHARD&collection=mycollection&shard=shard1
```

The output of the command is seen on the browser. We can see that **shard1** is split into two shards **shard1_0** and **shard1_1**:

While the query for shard splitting is being executed, the shard does not go offline. In fact, there is no interruption of service. Let us also look at the SolrCloud graph that would contain information on the split shards.

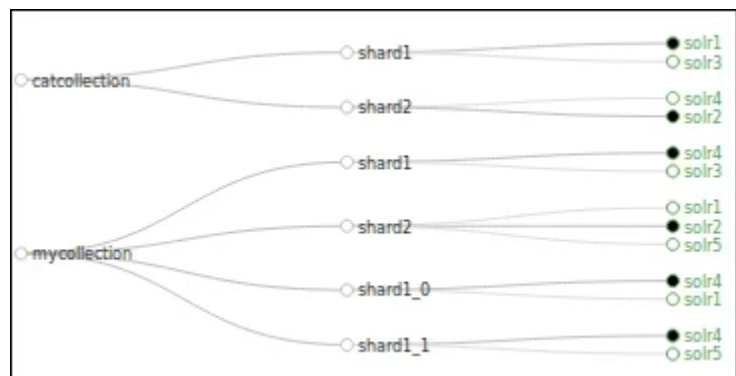
We can see that **shard1** has been split into **shard1_0** and **shard1_1**. Even the shards that have been split have their leaders and replicas in place. **Shard1_0** has **solr4** as the leader and **solr1** as the replica. Similarly, **shard1_1** has **solr4** as the leader and **solr5** as the replica. To check the number of documents in the split shards, execute the following queries:

```
http://solr1:8080/solr/mycollection/
q=*&shards=shard1_0
```

Shard1_0 contains eight documents.

```
http://solr1:8080/solr/mycollection/select/?q=*&shards=shard1_1
```

```
<?xml version="1.0"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">46981</int>
  </lst>
  <lst name="success">
    <lst>
      <lst name="responseHeader">
        <int name="status">0</int>
        <int name="QTime">12484</int>
      </lst>
      <str name="core">mycollection_shard1_1_replica1</str>
    </lst>
    <lst>
      <lst name="responseHeader">
        <int name="status">0</int>
        <int name="QTime">13421</int>
      </lst>
      <str name="core">mycollection_shard1_0_replica1</str>
    </lst>
  </lst>
</response>
```



Also, **shard1_1** contains nine documents. In all, the split shards now contain 17 documents that shard1 had earlier. In addition to splitting shard1 into two sub shards, SolrCloud makes the parent shard, shard1, inactive. This information is available in the ZooKeeper servers. Connect to any of the ZooKeeper servers and get the `clusterstate.json` file to check the status of the shards for `mycollection` :

```
"mycollection":{
  "shards":{
    "shard1":{
      "range":"80000000-ffffffff",
      "state":"inactive",
      "replicas":{
        "core_node1":{
          "state":"active",
          "base_url":"http://solr4:8080/solr",
          "core":"mycollection_shard1_replica1",
          "node_name":"solr4:8080_solr",
          "leader":"true"},
        "core_node4":{
          "state":"active",
          "base_url":"http://solr3:8080/solr",
          "core":"mycollection_shard1_replica2",
          "node_name":"solr3:8080_solr"}}},
```

Now, `shard1_0` and `shard1_1` are marked as active:


```

"shard1_0":{
  "range":"80000000-bfffffff",
  "state":"active",
  "replicas":{
    "core_node6":{
      "state":"active",
      "base_url":"http://solr4:8080/solr",
      "core":"mycollection_shard1_0_replica1",
      "node_name":"solr4:8080_solr",
      "leader":"true"},
    "core_node8":{
      "state":"active",
      "base_url":"http://solr1:8080/solr",
      "core":"mycollection_shard1_0_replica2",
      "node_name":"solr1:8080_solr"}}},
"shard1_1":{
  "range":"c0000000-ffffffff",
  "state":"active",
  "replicas":{
    "core_node7":{
      "state":"active",
      "base_url":"http://solr4:8080/solr",
      "core":"mycollection_shard1_1_replica1",
      "node_name":"solr4:8080_solr",
      "leader":"true"},
    "core_node9":{
      "state":"active",
      "base_url":"http://solr5:8080/solr",
      "core":"mycollection_shard1_1_replica2",
      "node_name":"solr5:8080_solr"}}}},

```

Deleting a shard

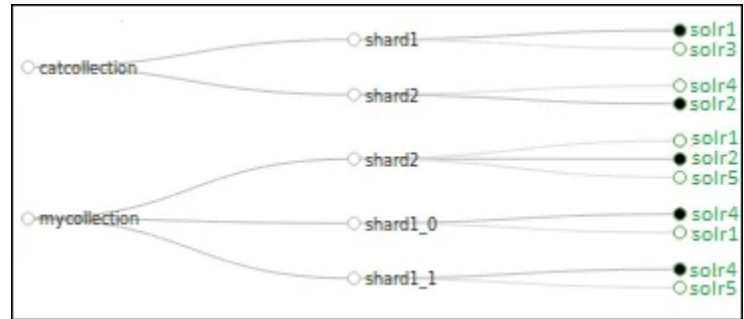
Only an inactive shard can be deleted. In the previous section, we found that since **shard1** was split into **shard1_0** and **shard1_1**, **shard1** was marked as inactive. We can delete shard1 by executing the **DELETESHARD** action on the collections API:

```

http://solr1:8080/solr/admin/collections?
action=DELETESHARD&collection=mycollection&shard=shard1

```

The following is a representation of the SolrCloud graph after the deletion of **shard1**:



Moving the existing shard to a new node

In order to move a shard to a new node, we need to add the node as a replica. Once the replication on the node is over and the node becomes active, we can simply shut down the old node and remove it from the cluster.

In the current cluster, we can see that **shard2** has three nodes—**solr1**, **solr2**, and **solr5**. We added **solr5** some time back as a replica for **shard2**. In order to remove **solr2** from **mycollection**, all we need to do is use the **DELETEREPLICA** action on the collections API:

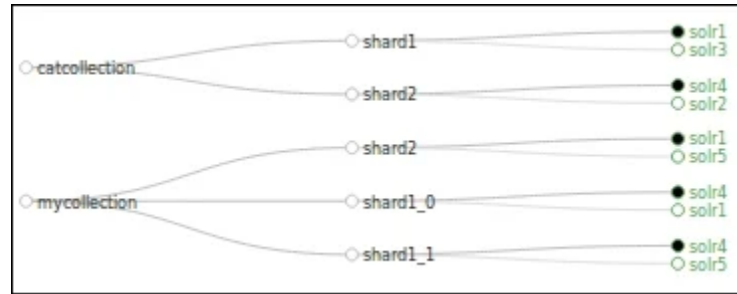
```
http://solr1:8080/solr/admin/collections?
action=DELETEREPLICA&collection=mycollection&shard=shard2&replica=core_node3
```

The name of the replica is obtained from **clusterstate.json** in the ZooKeeper cluster:

```
"mycollection":{
  "shards":{
    "shard2":{
      "range":"0-7fffffff",
      "state":"active",
      "replicas":{
        "core_node2":{
          "state":"active",
          "base_url":"http://solr1:8080/solr",
          "core":"mycollection_shard2_replica1",
          "node_name":"solr1:8080_solr",
          "leader":"true"},
        "core_node3":{
          "state":"active",
          "base_url":"http://solr2:8080/solr",
          "core":"mycollection_shard2_replica2",
          "node_name":"solr2:8080_solr"},
        "core_node5":{
          "state":"active",
          "base_url":"http://solr5:8080/solr",
          "core":"mycollection_shard2_replica3",
          "node_name":"solr5:8080_solr"}}},
  }
```

The graph now shows **solr2** is removed from **mycollection**:

In this case, **solr2**, which is also a node in **catcollection** is still active.



Shard splitting based on split key

Split key-based shard splitting is a viable option. A split key can be used to route documents on the basis of certain criteria to a shard in SolrCloud. In order to split a shard by using a shard key, we need to specify the **shard.key** parameter along with the collection parameter in the **SPLITSHARD** action of the collections API.

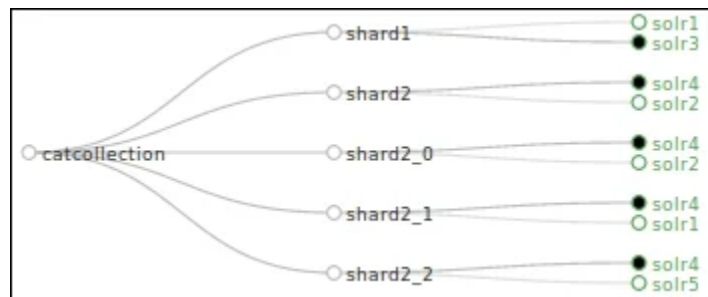
We can split **catcollection** into more shards using **category** as the **split.key** parameter. The URL for splitting the shard will be:

```
http://solr1:8080/solr/admin/collections?
action=SPLITSHARD&collection=catcollection&split.key=books!
```

Once the query has been executed, we can see the success message. It says that **Shard2** of **catcollection** has been broken into three shards, as shown in the following SolrCloud graph:

The complete process of splitting a shard into two and moving it to a separate new node in SolrCloud is required in the following scenarios:

- Average query performance on a shard or slowing down of a number of shards. It is important to measure this regularly and keep track of the number of queries per second.
- Degradation of indexing throughput. This is a scenario wherein you were able to index 1000 documents per second earlier, but it goes down to say 800 documents per second.
- Out of memory errors during querying. Even after tuning - query, cache and GC.





Asynchronous calls

Since some API calls, such as shard splitting, can take a long time and result in timeouts, we have the option of running a call asynchronously by specifying the `async=<request_id>` parameter in the URL. `<request_id>` is any ID that can be used to track the status of a particular API call. The `request_id` class and the status of the task are stored in ZooKeeper and can be retrieved using the `REQUESTSTATUS` action on the collections API.

Tip

We can delete and recreate `catcollection` using the following API calls or URLs:

```
http://solr1:8080/solr/admin/collections?action=DELETE&name=catcollection
http://solr1:8080/solr/admin/collections?
action=CREATE&name=catcollection&numShards=2&replicationFactor=2&maxShardsPerNode=2&
```

`catcollection` can be populated using `docs.csv` file provided by the following command:

```
java -Dtype=text/csv -Durl=http://solr4:8080/solr/catcollection/update -jar post.jar
docs.csv
```

To perform splitting using the `async` parameter, execute the following command:

```
http://solr1:8080/solr/admin/collections?
action=SPLITSHARD&collection=catcollection&split.key=books!&async=1111
```

We immediately get a response which just shows the `requestid` that we submitted. The status of the request can be checked by executing the following URL:

```
http://solr1:8080/solr/admin/collections?
action=REQUESTSTATUS&requestid=1111
```

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">106</int>
  </lst>
  <str name="requestid">1111</str>
</response>
```

Here we can see that the status is marked as completed. These requests and their status are stored in ZooKeeper and are not cleaned up automatically. We can clean up the requests by passing `requestid` as `-1` :

```
http://solr1:8080/solr/admin/collections?action=REQUESTSTATUS&requestid=-1
```



```
<?xml version="1.0"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">19</int>
  </lst>
  <lst name="status">
    <str name="state">completed</str>
    <str name="msg">found 1111 in completed tasks</str>
  </lst>
</response>
```

Migrating documents to another collection

Suppose we have a huge collection of over a billion documents and we get a requirement whereby we need to create a separate index with a particular set of documents, or we want to break our index into two parts on the basis of certain criteria. Migration of documents to another collection makes this possible. Effectively, we can specify a source and a destination collection in SolrCloud. On the basis of the routing criteria, certain documents will be copied from the source to the destination collection. We can specify the migration time as the `forward.timeout` parameter during which all write requests will be forwarded to the target collection. The target collection must not receive any writes while the migrate command is running. Otherwise, some writes may be lost.

Let us look at a practical scenario.

We currently have two collections— `catcollection` and `mycollection`. Now `catcollection` contains documents belonging to the categories `books`, `currency`, and `electronics`. Let us move the documents belonging to the category `currency` from `catcollection` to `mycollection`.

The query to get the documents belonging to category `currency` will include the `shard.keys=currency!` parameter:

```
http://solr1:8080/solr/catcollection/select/?q=*&rows=15&shard.keys=currency!
```

We can see that there are 4 documents in the collection. On querying the `mycollection` collection, we find that there are 35 documents in the collection. Now, let us copy the documents from `catcollection` to `mycollection`:

```
http://solr1:8080/solr/admin/collections?
action=MIGRATE&collection=catcollection&split.key=currency!&target.collection=mycollection
```

Note the following:

- The action is `MIGRATE`.
- The source collection is `catcollection`.

- `split.key` is `currency!` . All documents that have `currency!*` as the ID will be moved to `mycollection` . `split.key` is identified by the routing parameter that we used earlier. If there is no routing parameter, `split.key` can be identified by the unique ID of the documents.
- `target.collection` refers to the target `mycollection` .
- `forward.timeout` is the timeout specified during which all write requests to `catcollection` are forwarded to `mycollection` .

A success message is displayed once this completes.

We can see the routing parameters in the `clusterstate.json` file. This also includes an `expiresAt` parameter specifying the time after which the forwarding of requests to the target collection is stopped:

Once the migration is over, the destination collection, `mycollection` , will contain 4 more documents, with the number totaling to 39. These documents will also be available in the source collection.

```
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">28836</int>
  </lst>
  <lst name="success">
    <lst>
      <lst name="responseHeader">
        <int name="status">0</int>
        <int name="QTime">2</int>
      </lst>
      <str name="core">mycollection shard1_0_replica2</str>
      <str name="status">BUFFERING</str>
    </lst>
  </lst>
```

```
"routingRules":{"currency!":{"routeRanges":"92a40000-92a4ffff",
"expireAt":"1404265881748",
"targetCollection":"mycollection"}}},
```

Tip

Solr collections API reference:

<https://cwiki.apache.org/confluence/display/solr/Collections+API>.



Sizing and monitoring of SolrCloud

It is important to understand that SolrCloud is horizontally scalable. However, each node needs to have a certain capacity. The amount of CPU, disk, and RAM required for each node in SolrCloud needs to be figured out for the efficient allocation of resources. Though no fixed number can be assigned to these parameters as each application is unique, each index within each application has a unique indexing pattern—the number of documents that need to be indexed per second, the size of the documents, and the fields, the tokenization and the storage parameters defined. Similarly, the search patterns would also differ across indexes belonging to different applications. The number of queries per second and the search parameters can be different. The amount of data retrieved from Solr and the faceting and grouping parameters play an important role in the handling of resources used during querying.

Therefore, it is difficult to assign numbers to the RAM, CPU, and disk requirements for each node. Ideally, we should implement sharding on the size of the shard instead of the size of the collection. Routing is another very important parameter in the index. It would save a lot of network IO. The weight of a particular shard depends on the routing parameter, either in terms of the number of documents or the number of queries per second.

It is important to restrict the disk space to two to three times the size of the index. When index optimization happens, it uses up more than twice the disk space.

The ideal way to go about sizing is to put a few normal machines as the nodes of SolrCloud and monitor their resource usage. For each node, we need to monitor the following parameters:

- Load average or CPU usage
- Disk usage
- RAM usage, or RAM utilization across each core
- Core to CPU consumption, or CPU utilization across each core
- Collection to node consumption, or how the requests are being distributed across each node in the collection

Once these parameters are in place, nodes belonging to some shards are found to be overweight. These shards need to be split further in order to properly address scalability issues that may occur in the future.

The health of SolrCloud can be monitored via the following files or directories in the ZooKeeper server:

- `clusterstate.json`

- / **livenodes**

We need to constantly watch the state of each core in the cluster. **livenodes** provides us with a list of available nodes. If any node or core goes offline, a notification has to be sent out. Additionally, it is important to have enough replicas distributed in such a fashion that a core or a node going down should not affect the availability of the cloud. The following points need to be considered while planning out the nodes, cores, and replicas of SolrCloud:

- Each collection should have an appropriate number of shards
- Shards should have a leader and more than one replica
- Leaders and replicas should be on different physical nodes
- Even when using virtual machines, the third step should be considered
- A few standby nodes, which can be assigned as replicas, should be set up, if needed
- An automated process should be followed for setting up standby nodes
- An automated process should be followed for spawning new nodes
- Checks on the network IO should be scheduled to identify the network or cluster traffic and continually optimized

These action points will make the SolrCloud function in an effortless manner.



Using SolrCloud as a NoSQL database

There is a huge market for NoSQL databases, each having its own strength and weakness. Several factors need consideration during the selection of a NoSQL database, namely performance, scalability, security, and ease of development. RDBMS is good but has limitations in terms of scaling to billions of records. Horizontal scaling is a challenge in most RDBMSs.

Search, which was earlier a complex process, is now easy to use and scale. With horizontal scalability, search has also become affordable. NoSQL databases can be key-value, column oriented, document oriented and graph database. The key factors that are used to make a

decision regarding the NoSQL database are as follows:

- **Data model:** Refers to how data is stored and accessed or whether the NoSQL database is key-value, document oriented, or column oriented.
- **Distribution model:** Refers to how data is distributed across the cluster to address horizontal scalability. It considers sharding and replication features.
- **Conflict resolution:** Refers to how data is kept consistent across the nodes in the cluster. It ensures that all the nodes apply the operations in the same order and takes care of update and read consistency.

Each NoSQL database needs a search option. MongoDB, Redis, CouchDB, Riak, and other NoSQL databases provide search options, though the search is not as effectively implemented in these databases. Ideally, we need to come down to using Lucene, Solr, or Elasticsearch.

Therefore, instead of adding the search feature to the database, we can add the database to the search function. This means that we can store data inside the Solr index and retrieve it during search. Solr is a document-oriented data store, which is closer to the MongoDB data model. With the latest version of Solr and near real-time functionalities, we obtain the following features:

- Real-time get
- Update durability
- Atomic compare and set
- Versioning and optimistic locking

This brings Solr closer to being a NoSQL database. Talking about the schema less - Solr is effectively '*schemaless*'. We need not run a lengthy alter command to add new fields to the schema. The schema can be altered and new fields indexed in the new schema without affecting the documents that are already present in the index. Altering the schema of existing fields can cause problems. In that case, those fields would need to be re-indexed. However, adding new fields to the index does not affect the existing documents in any way. Any search on those fields would simply ignore the documents in which the fields do not exist.

SolrCloud takes this further by providing horizontal scalability to the Solr database. Solr is 'eventually consistent'.

Eventual consistency is a consistency model used in distributed computing that informally guarantees that, if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value.

Source: http://en.wikipedia.org/wiki/Eventual_consistency.

The duration of the presence of inconsistency is known as the inconsistency window. SolrCloud has a very small inconsistency window that depends on the size of the data, the command to be executed, and the network.

This effectively means that Solr can be used as a NoSQL database to store, search, and retrieve data.



Summary

In this chapter, we went through most of the aspects of SolrCloud. We understood the architecture of SolrCloud, constructed a setup for SolrCloud using ZooKeeper servers, and created our collections on the cloud. We saw the advantages of routing and how to implement it. We saw how SolrCloud addresses the horizontal scalability, high availability, and distributed indexing and search requirements for a large-scale Solr deployment. We saw how to manage the shards in SolrCloud and monitor SolrCloud. We can use the monitoring information to size the cores in SolrCloud and scale it further. We also saw that SolrCloud can be used as a NoSQL database.

In the next chapter, we will explore text tagging using the Lucene **Finite State Transducer (FST)**. We will delve into FSTs and how they can be implemented using Lucene and SolrCloud?



